Java (programming language)

From Wikipedia, the free encyclopedia

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun made available most of their Java technologies as free software under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

Java

Paradigm	Object-oriented, structured, imperative
Appeared in	1995
Designed by	Sun Microsystems
Latest release	Java Standard Edition 6 (1.6.13)
Typing discipline	Static, strong, safe, nominative, manifest
Major implementations	Numerous
Influenced by	Ada 83, C++, C#,[1] Eiffel,[2] Smalltalk, Mesa,[3] Modula-3,[4] Objective-C
Influenced	Ada 2005, C#, D, ECMAScript, Groovy, J#, PHP, Scala, JavaScript
OS	Cross-platform (multi-platform)
License	GNU General Public License / Java Community Process
Website	http://java.sun.com

Contents

History

See also: Java (Sun) history and Java version history

Duke, the Java mascot

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects.[5] The language, initially called *Oak* after an oak tree that stood outside Gosling's office, also went by the name *Green* and ended up later renamed as *Java*, from a list of random words.[6] Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation.[7]

Sun released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access restrictions. Major web browsers soon incorporated the ability to run Java *applets* within web pages, and Java quickly became popular. With the advent of *Java 2* (released initially as J2SE 1.2 in December 1998), new versions had multiple configurations built for different types of platforms. For example, *J2EE* targeted enterprise applications and the greatly stripped-down version *J2ME* for mobile applications. *J2SE* designated the Standard Edition. In 2006, for marketing purposes, Sun renamed new *J2* versions as *Java EE*, *Java ME*, and *Java SE*, respectively.

In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process.[8] Java remains a *de facto* standard, controlled through the Java Community Process.[9] At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) (a subset of the SDK); the primary distinction involves the JRE's lack of the compiler, utility programs, and header files.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL). On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.[10]

Philosophy

Primary goals

There were five primary goals in the creation of the Java language:[11]

1. It should be "simple, object oriented, and familiar".
2. It should be "robust and secure".
3. It should be "architecture neutral and portable".
4. It should execute with "high performance".
5. It should be "interpreted, threaded, and dynamic".

Java Platform

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. One should be able to write a program once, compile it once, and run it anywhere.

This is achieved by compiling the Java language code, not to machine code but to Java bytecode – instructions analogous to machine code but intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

Standardized libraries provide a generic way to access host specific features such as graphics, threading and networking. In some JVM versions, bytecode can be compiled to native code, either before or during program execution, resulting in faster execution.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would, and Java suffered a reputation for poor performance. This gap has been narrowed by a number of optimization techniques introduced in the more recent JVM implementations.

One such technique, known as just-in-time (JIT) compilation, translates Java bytecode into native code the first time that code is executed, then caches it. This results in a program that starts and executes faster than pure interpreted code can, at the cost of introducing occasional compilation overhead during execution. More sophisticated VMs also use dynamic recompilation, in which the VM analyzes the behavior of the running program and selectively recompiles and optimizes parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes, and can identify *hot spots* - parts of the program, often inner loops, that take up the most execution time. JIT compilation and dynamic recompilation allow Java programs to approach the speed of native code without losing portability.

Another technique, commonly known as *static compilation*, or ahead-of-time (AOT) compilation, is to compile directly into native code like a more traditional compiler. Static Java compilers translate the Java source or bytecode to native object code. This achieves good performance compared to interpretation, at the expense of portability; the output of these compilers can only be run on a single architecture. AOT could give Java something close to native performance, yet it is still not portable since there are no compiler directives, and all the pointers are indirect with no way to micro manage garbage collection.

Java's performance has improved substantially since the early versions, and performance of JIT compilers relative to native compilers has in some tests been shown to be quite similar.[12][13] The performance of the compilers does not necessarily indicate the performance of the compiled code; only careful testing can reveal the true performance issues in any system.

One of the unique advantages of the concept of a runtime engine is that even the most serious errors (exceptions) in a Java program should not 'crash' the system under any circumstances, provided the JVM itself is properly implemented. Moreover, in runtime engine environments such as Java there exist tools that attach to the runtime engine and every time that an exception of interest occurs they record debugging information that existed in memory at the time the exception was thrown (stack and heap values). These Automated Exception Handling tools provide 'root-cause' information for exceptions in Java programs that run in production, testing or development environments. Such precise debugging is much more difficult to implement without the run-time support that the JVM offers.

Implementations

Sun Microsystems officially licenses the Java Standard Edition platform for Microsoft Windows, Linux, Mac OS X, and Solaris. Through a network of third-party vendors and licensees[14], alternative Java environments are available for these and other platforms.

Sun's trademark license for usage of the Java brand insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support RMI or JNI and had added platform-specific features of their own. Sun sued in 1997, and in 2001 won a settlement of $20 million as well as a court order enforcing the terms of the license from Sun.[15] As a result, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support Java applets without a third-party plugin. Sun, and others, have made available free Java run-time systems for those and other versions of Windows.

Platform-independent Java is essential to the Java EE strategy, and an even more rigorous validation is required to certify an implementation. This environment enables portable server-side applications, such as Web services, servlets, and Enterprise JavaBeans, as well as with embedded systems based on OSGi, using Embedded Java environments. Through the new GlassFish project, Sun is working to create a fully functional, unified open-source implementation of the Java EE technologies.

Sun also distributes a superset of the JRE called the Java 2 SDK (more commonly known as the JDK), which includes development tools such as the Java compiler, Javadoc, Jar and debugger.

Automatic memory management

See also: Garbage collection (computer science)

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable object becomes eligible to be freed automatically by the garbage collector. Something similar to a memory leak may still occur if a programmer's code holds a reference to an object that is no longer needed, typically when objects that are no longer needed are stored in containers that are still in use. If methods for a nonexistent object are called a "null pointer exception" is thrown. [16][17]

One of the ideas behind Java's automatic memory management model is that programmers be spared the burden of having to perform manual memory management. In some languages memory for the creation of objects is implicitly allocated on the stack, or explicitly allocated and deallocated from the heap. Either way the responsibility of managing memory resides with the programmer. If the program does not deallocate an object, a memory leak occurs. If the program attempts to access or deallocate memory that has already been deallocated, the result is undefined and difficult to predict, and the program is likely to become unstable and/or crash. This can be partially remedied by the use of smart pointers, but these add overhead and complexity.

Garbage collection may happen at any time. Ideally, it will occur when a program is idle. It is guaranteed to be triggered if there is insufficient free memory on the heap to allocate a new object; this can cause a program to stall momentarily. Where performance or response time is important, explicit memory management and object pools are often used.

Java does not support C/C++ style pointer arithmetic, where object addresses and unsigned integers (usually long integers) can be used interchangeably. This allows the garbage collector to relocate referenced objects, and ensures type safety and security.

As in C++ and some other object-oriented languages, variables of Java's primitive types are not objects. Values of primitive types are either stored directly in fields (for objects) or on the stack (for methods) rather than on the heap, as commonly true for objects (but see Escape analysis). This was a conscious decision by Java's designers for performance reasons. Because of this, Java was not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to proceed as if primitive types are instances of their wrapper classes.

Syntax

The syntax of Java is largely derived from C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object oriented language. All code is written inside a class and everything is an object, with the exception of the intrinsic data types (ordinal and real numbers, boolean values, and characters), which are not classes for performance reasons.

Java suppresses several features (such as operator overloading and multiple inheritance) for *classes* in order to simplify the language and to prevent possible errors and anti-pattern design.

Examples

Hello world

The traditional Hello world program can be written in Java as:

```
/*
 * Outputs "Hello, World!" and then exits
 */

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

By convention, source files are named after the public class they contain, appending the suffix *.java*, for example, `HelloWorld.java`. It must first be compiled into bytecode, using a Java compiler, producing a file named `HelloWorld.class`. Only then can it be executed, or 'launched'. The java source file may only contain one public class but can contain multiple classes with less than public access and any number of public inner classes.

A `class` that is declared `private` may be stored in any .java file. The compiler will generate a class file for each class defined in the source file. The name of the class file is the name of the class, with *.class* appended. For class file generation, anonymous classes are treated as if their name was the concatenation of the name of their enclosing class, a *$*, and an integer.

The keyword `public` denotes that a method can be called from code in other classes, or that a class may be used by classes outside the class hierarchy. The class hierarchy is related to the name of the directory in which the.java file is.

The keyword `static` in front of a method indicates a static method, which is associated only with the class and not with any specific instance of that class. Only static methods can be invoked without a reference to an object. Static methods cannot access any method variables that are not static.

The keyword `void` indicates that the main method does not return any value to the caller. If a Java program is to exit with an error code, it must call System.exit()

The method name "`main`" is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program. Java classes that run in managed environments such as applets and Enterprise Java Beans do not use or need a `main()` method. A java program may contain multiple classes that have `main` methods, which means that the VM needs to be explicitly told which class to launch from.

The main method must accept an array of `String` (`http://java.sun.com/javase/6/docs/api/java/lang/String.html`) objects. By convention, it is referenced as `args` although any other legal identifier name can be used. Since Java 5, the main method can also use variable arguments, in the form of `public static void main(String... args)`, allowing the main method to be invoked with an arbitrary number of `String` arguments. The effect of this alternate declaration is semantically identical (the `args` parameter is still an array of `String` objects), but allows an alternate syntax for creating and passing the array.

The Java launcher launches Java by loading a given class (specified on the command line or as an attribute in a JAR)

and starting its `public static void main(String[])` method. Stand-alone programs must declare this method explicitly. The `String[] args` parameter is an array of `String` (http://java.sun.com/javase/6/docs/api/java/lang/String.html) objects containing any arguments passed to the class. The parameters to `main` are often passed by means of a command line.

Printing is part of a Java standard library: The **`System`** **(http://java.sun.com/javase/6/docs/api/java/lang/System.html)** class defines a public static field called **`out`** **(http://java.sun.com/javase/6/docs/api/java/lang/System.html#out)**. The `out` object is an instance of the `PrintStream` (http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html) class and provides many methods for printing data to standard out, including **`println(String)`** **(http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html#println(java.lang.String))** which also appends a new line to the passed string.

The string "Hello world!" is automatically converted to a String object by the compiler.

A more comprehensive example

```
// OddEven.java
import javax.swing.JOptionPane;

public class OddEven {
    // "input" is the number that the user gives to the computer
    private int input; // a whole number("int" means integer)

    /*
     * This is the constructor method. It gets called when an object of the OddEven type
     * is being created.
     */
    public OddEven() {
    //Code not shown
    }

    // This is the main method. It gets called when this class is run through a Java interpreter.
    public static void main(String[] args) {
        /*
         * This line of code creates a new instance of this class called "number" (also known as an
         * Object) and initializes it by calling the constructor.  The next line of code calls
         * the "showDialog()" method, which brings up a prompt to ask you for a number
         */
        OddEven number = new OddEven();
        number.showDialog();
    }

    public void showDialog() {
        /*
         * "try" makes sure nothing goes wrong. If something does,
         * the interpreter skips to "catch" to see what it should do.
         */
        try {
            /*
             * The code below brings up a JOptionPane, which is a dialog box
             * The String returned by the "showInputDialog()" method is converted into
             * an integer, making the program treat it as a number instead of a word.
             * After that, this method calls a second method, calculate() that will
             * display either "Even" or "Odd."
             */
            input = new Integer(JOptionPane.showInputDialog("Please Enter A Number"));
            calculate();
        } catch (NumberFormatException e) {
            /*
             * Getting in the catch block means that there was a problem with the format of
             * the number. Probably some letters were typed in instead of a number.
             */
            System.err.println("ERROR: Invalid input. Please type in a numerical value.");
        }
    }

    /*
     * When this gets called, it sends a message to the interpreter.
     * The interpreter usually shows it on the command prompt (For Windows users)
```

```
     * or the terminal (For Linux users). (Assuming it's open)
     */
    private void calculate() {
        if (input % 2 == 0) {
            System.out.println("Even");
        } else {
            System.out.println("Odd");
        }
    }
}
```

- The **import** statement imports the `JOptionPane` (`http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html`) class from the `javax.swing` (`http://java.sun.com/javase/6/docs/api/javax/swing/package-summary.html`) package.
- The `OddEven` class declares a single `private` field of type `int` named `input`. Every instance of the `OddEven` class has its own copy of the `input` field. The private declaration means that no other class can access (read or write) the `input` field.
- `OddEven()` is a `public` constructor. Constructors have the same name as the enclosing class they are declared in, and unlike a method, have no return type. A constructor is used to initialize an object that is a newly created instance of the class.
- The dialog returns a `String` that is converted to an `int` by the `Integer.parseInt(String)` (`http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#parseInt(String)`) method.
- The `calculate()` method is declared without the `static` keyword. This means that the method is invoked using a specific instance of the `OddEven` class. (The reference used to invoke the method is passed as an undeclared parameter of type `OddEven` named **`this`**.) The method tests the expression `input % 2 == 0` using the **`if`** keyword to see if the remainder of dividing the `input` field belonging to the instance of the class by two is zero. If this expression is true, then it prints **Even**; if this expression is false it prints **Odd**. (The `input` field can be equivalently accessed as `this.input`, which explicitly uses the undeclared `this` parameter.)
- `OddEven number = new OddEven();` declares a local object reference variable in the `main` method named `number`. This variable can hold a reference to an object of type `OddEven`. The declaration initializes `number` by first creating an instance of the `OddEven` class, using the **`new`** keyword and the `OddEven()` constructor, and then assigning this instance to the variable.
- The statement `number.showDialog();` calls the calculate method. The instance of `OddEven` object referenced by the `number` local variable is used to invoke the method and passed as the undeclared `this` parameter to the `calculate` method.
- `input = new Integer(JOptionPane.showInputDialog("Please Enter A Number"));` is a statement that converts the type of **String** to the primitive type **int** by taking advantage of the wrapper class **Integer**.

Special Classes

Applet

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

The `import` statements direct the Java compiler to include the `javax.swing.JApplet`

(http://java.sun.com/javase/6/docs/api/javax/swing/JApplet.html) and java.awt.Graphics (http://java.sun.com/javase/6/docs/api/java/awt/Graphics.html) classes in the compilation. The import statement allows these classes to be referenced in the source code using the *simple class name* (i.e. JApplet) instead of the *fully qualified class name* (i.e. javax.swing.JApplet).

The Hello class extends (subclasses) the JApplet (Java Applet) class; the JApplet class provides the framework for the host application to display and control the lifecycle of the applet. The JApplet class is a JComponent (Java Graphical Component) which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The Hello class overrides the paintComponent(Graphics) (http://java.sun.com/javase/6/docs/api/java/awt/Container.html#paint(java.awt.Graphics)) method inherited from the Container (http://java.sun.com/javase/6/docs/api/java/awt/Container.html) superclass to provide the code to display the applet. The paint() method is passed a Graphics object that contains the graphic context used to display the applet. The paintComponent() method calls the graphic context drawString (String, int, int) (http://java.sun.com/javase/6/docs/api/java/awt/Graphics.html#drawString(java.lang.String,%20int,%20int)) method to display the **"Hello, world!"** string at a pixel offset of (65, 95) from the upper-left corner in the applet's display.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!-- Hello.html -->
<html>
  <head>
    <title>Hello World Applet</title>
  </head>
  <body>
    <applet code="Hello" width="200" height="200">
    </applet>
  </body>
</html>
```

An applet is placed in an HTML document using the <applet> HTML element. The applet tag has three attributes set: code="Hello" specifies the name of the JApplet class and width="200" height="200" sets the pixel width and height of the applet. Applets may also be embedded in HTML using either the object or embed element[18], although support for these elements by Web browsers is inconsistent.[19] However, the applet tag is deprecated, so the object tag is preferred where supported.

The host application, typically a Web browser, instantiates the Hello applet and creates an AppletContext (http://java.sun.com/javase/6/docs/api/java/applet/AppletContext.html) for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The paint method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

Servlet

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. Servlets are server-side Java EE components that generate responses (typically HTML pages) to requests (typically HTTP requests) from clients. A servlet can almost be thought of as an applet that runs on the server side—without a face.

```
// Hello.java
import java.io.*;
import javax.servlet.*;

public class Hello extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response)
```

```
        throws ServletException, IOException {
    response.setContentType("text/html");
    final PrintWriter pw = response.getWriter();
    pw.println("Hello, world!");
    pw.close();
  }
}
```

The `import` statements direct the Java compiler to include all of the public classes and interfaces from the `java.io` (`http://java.sun.com/javase/6/docs/api/java/io/package-summary.html`) and `javax.servlet` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/package-summary.html`) packages in the compilation.

The `Hello` class `extends` the `GenericServlet` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/GenericServlet.html`) class; the `GenericServlet` class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.

The `Hello` class overrides the `service(ServletRequest, ServletResponse)` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/Servlet.html#service(javax.servlet.ServletRequest,javax.servlet.ServletResponse)`) method defined by the `Servlet` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/Servlet.html`) interface to provide the code for the service request handler. The `service()` method is passed a `ServletRequest` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletRequest.html`) object that contains the request from the client and a `ServletResponse` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletResponse.html`) object used to create the response returned to the client. The `service()` method declares that it `throws` the exceptions `ServletException` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletException.html`) and `IOException` (`http://java.sun.com/javase/6/docs/api/java/io/IOException.html`) if a problem prevents it from responding to the request.

The `setContentType(String)` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletResponse.html#setContentType(java.lang.String)`) method in the response object is called to set the MIME content type of the returned data to **"text/html"**. The `getWriter()` (`http://java.sun.com/javaee/5/docs/api/javax/servlet/ServletResponse.html#getWriter()`) method in the response returns a `PrintWriter` (`http://java.sun.com/javase/6/docs/api/java/io/PrintWriter.html`) object that is used to write the data that is sent to the client. The `println(String)` (`http://java.sun.com/javase/6/docs/api/java/io/PrintWriter.html#println(java.lang.String)`) method is called to write the **"Hello, world!"** string to the response and then the `close()` (`http://java.sun.com/javase/6/docs/api/java/io/PrintWriter.html#close()`) method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.

JavaServer Page

JavaServer Pages (JSPs) are server-side Java EE components that generate responses, typically HTML pages, to HTTP requests from clients. JSPs embed Java code in an HTML page by using the special delimiters `<%` and `%>`. A JSP is compiled to a Java *servlet*, a Java application in its own right, the first time it is accessed. After that, the generated servlet creates the response.

Swing application

Swing is a graphical user interface library for the Java SE platform. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK and Motif are supplied by Sun. Apple

also provides an Aqua look and feel for Mac OS X. Where prior implementations of these looks and feels may have been considered lacking, Swing in Java SE 6 addresses this problem by using more native widget drawing routines of the underlying platforms.

This example Swing application creates a single window with "Hello, world!" inside:

```
// Hello.java (Java SE 5)
import java.awt.BorderLayout;
import javax.swing.*;

public class Hello extends JFrame {
    public Hello() {
        super("hello");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        add(new JLabel("Hello, world!"));
        pack();
    }

    public static void main(String[] args) {
        new Hello().setVisible(true);
    }
}
```

The first `import` statement directs the Java compiler to include the `BorderLayout` (http://java.sun.com/javase/6/docs/api/java/awt/BorderLayout.html) class from the `java.awt` (http://java.sun.com/javase/6/docs/api/java/awt/package-summary.html) package in the compilation; the second `import` includes all of the public classes and interfaces from the `javax.swing` (http://java.sun.com/javase/6/docs/api/javax/swing/package-summary.html) package.

The `Hello` class `extends` the `JFrame` (http://java.sun.com/javase/6/docs/api/javax/swing/JFrame.html) class; the `JFrame` class implements a window with a title bar and a close control.

The `Hello()` constructor initializes the frame by first calling the superclass constructor, passing the parameter `"hello"`, which is used as the window's title. It then calls the `setDefaultCloseOperation(int)` (http://java.sun.com/javase/6/docs/api/javax/swing/JFrame.html#setDefaultCloseOperation(int)) method inherited from `JFrame` to set the default operation when the close control on the title bar is selected to `WindowConstants.EXIT_ON_CLOSE` (http://java.sun.com/javase/6/docs/api/javax/swing/WindowConstants.html#EXIT_ON_CLOSE) — this causes the `JFrame` to be disposed of when the frame is closed (as opposed to merely hidden), which allows the JVM to exit and the program to terminate. Next, the layout of the frame is set to a `BorderLayout`; this tells Swing how to arrange the components that will be added to the frame. A `JLabel` (http://java.sun.com/javase/6/docs/api/javax/swing/JLabel.html) is created for the string **"Hello, world!"** and the `add(Component)` (http://java.sun.com/javase/6/docs/api/java/awt/Container.html#add(java.awt.Component)) method inherited from the `Container` (http://java.sun.com/javase/6/docs/api/java/awt/Container.html) superclass is called to add the label to the frame. The `pack()` (http://java.sun.com/javase/6/docs/api/java/awt/Window.html#pack()) method inherited from the `Window` (http://java.sun.com/javase/6/docs/api/java/awt/Window.html) superclass is called to size the window and lay out its contents, in the manner indicated by the `BorderLayout`.

The `main()` method is called by the JVM when the program starts. It instantiates a new `Hello` frame and causes it to be displayed by calling the `setVisible(boolean)` (http://java.sun.com/javase/6/docs/api/java/awt/Component.html#setVisible(boolean)) method inherited from the `Component` (http://java.sun.com/javase/6/docs/api/java/awt/Component.html) superclass with the boolean parameter `true`. Once the frame is displayed, exiting the `main` method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing top-level

windows have been disposed.

Generics

Prior to generics, each variable declaration had to be of a specific type. For container classes, for example, this is a problem because there is no easy way to create a container that accepts only specific types of objects. Either the container operates on all subtypes of a class or interface, usually `Object`, or a different container class has to be created for each contained class. Generics allow compile-time type checking without having to create a large number of container classes, each containing almost identical code.

Class libraries

- Java libraries are the compiled byte codes of source code developed by the JRE implementor to support application development in Java. Examples of these libraries are:
  - The core libraries, which include:
    - Collection libraries that implement data structures such as lists, dictionaries, trees and sets
    - XML Processing (Parsing, Transforming, Validating) libraries
    - Security
    - Internationalization and localization libraries
  - The integration libraries, which allow the application writer to communicate with external systems. These libraries include:
    - The Java Database Connectivity (JDBC) API for database access
    - Java Naming and Directory Interface (JNDI) for lookup and discovery
    - RMI and CORBA for distributed application development
    - JMX for managing and monitoring applications
  - User Interface libraries, which include:
    - The (heavyweight, or native) Abstract Window Toolkit (AWT), which provides GUI components, the means for laying out those components and the means for handling events from those components
    - The (lightweight) Swing libraries, which are built on AWT but provide (non-native) implementations of the AWT widgetry
    - APIs for audio capture, processing, and playback
- A platform dependent implementation of Java virtual machine (JVM) that is the means by which the byte codes of the Java libraries and third party applications are executed
- Plugins, which enable applets to be run in Web browsers
- Java Web Start, which allows Java applications to be efficiently distributed to end users across the Internet
- Licensing and documentation.

Documentation

Javadoc is a comprehensive documentation system, created by Sun Microsystems, used by many Java developers. It provides developers with an organized system for documenting their code. Whereas normal comments in Java and C are set off with /* and */, the multi-line comment tags, Javadoc comments have an extra asterisk at the beginning, so that the tags are /** and */.

Examples

The following is an example of java code commented with simple Javadoc-style comments:

```
/**
 * A program that does useful things.
 */
```

```
public class Program {

  /**
   * A main method.
   * @param args The arguments
   */
  public static void main(String[] args) {
    //do stuff
  }

}
```

Editions

See also: Free Java implementations#Class library

Sun has defined and supports four editions of Java targeting different application environments and segmented many of its APIs so that they belong to one of the platforms. The platforms are:

- Java Card for smartcards
- Java Platform, Micro Edition (Java ME) — targeting environments with limited resources.
- Java Platform, Standard Edition (Java SE) — targeting workstation environments.
- Java Platform, Enterprise Edition (Java EE) — targeting large distributed enterprise or Internet environments.

The classes in the Java APIs are organized into separate groups called packages. Each package contains a set of related interfaces, classes and exceptions. Refer to the separate platforms for a description of the packages available.

Java editions

Java Card

Micro Edition (ME)

Standard Edition (SE)

Enterprise Edition (EE)

PersonalJava (discontinued)

The set of APIs is controlled by Sun Microsystems in cooperation with others through the Java Community Process program. Companies or individuals participating in this process can influence the design and development of the APIs. This process has been a subject of controversy.

Sun also provided an edition called PersonalJava that has been superseded by later, standards-based Java ME configuration-profile pairings.

Criticism

See also

- Comparison of programming languages
- Comparison of Java and C++
- Comparison of Java and C#
- JavaOne
- Javapedia
- List of Java virtual machines
- List of Java APIs
- List of JVM languages
- List of Java scripting languages
- Java version history

Notes

1. ^ Java 5.0 added several new language features (the enhanced for loop, autoboxing, varargs and annotations), after they were introduced in the similar (and competing) C# language. [1] (http://www.barrycornelius.com/papers/java5/)[2] (http://www.levenez.com/lang/)
2. ^ "The Java Language Environment" (http://java.sun.com/docs/white/langenv/Intro.doc1.html#943). May 1996. http://java.sun.com/docs/white/langenv/Intro.doc1.html#943.
3. ^ "The Java Language Specification, 2nd Edition" (http://java.sun.com/docs/books/jls/second_edition/html/intro.doc.html#237601). http://java.sun.com/docs/books/jls/second_edition/html/intro.doc.html#237601.
4. ^ http://www.computerworld.com.au/index.php/id;1422447371;pp;3;fp;4194304;fpid;1
5. ^ Jon Byous, *Java technology: The early years* (http://java.sun.com/features/1998/05/birthday.html). Sun Developer Network, no date [ca. 1998]. Retrieved April 22, 2005.
6. ^ http://blogs.sun.com/jonathan/entry/better_is_always_different.
7. ^ Heinz Kabutz, *Once Upon an Oak* (http://www.artima.com/weblogs/viewpost.jsp?thread=7555). Artima, Retrieved April 29, 2007.
8. ^ Java Study Group (http://www.open-std.org/JTC1/SC22/JSG/); Why Java Was - Not - Standardized Twice (http://csdl2.computer.org/comp/proceedings/hicss/2001/0981/05/09815015.pdf); What is ECMA—and why Microsoft cares (http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2832719,00.html)
9. ^ Java Community Process website (http://www.jcp.org/en/home/index)
10. ^ open.itworld.com - JAVAONE: Sun - The bulk of Java is open sourced (http://open.itworld.com/4915/070508opsjava/page_1.html)
11. ^ 1.2 Design Goals of the JavaTM Programming Language (http://java.sun.com/docs/white/langenv/Intro.doc2.html)
12. ^ Performance of Java versus C++ (http://www.idiom.com/~zilla/Computer/javaCbenchmark.html), J.P.Lewis and Ulrich Neumann, Computer Graphics and Immersive Technology Lab, University of Southern California
13. ^ FreeTTS - A Performance Case Study (http://research.sun.com/techrep/2002/smli_tr-2002-114.pdf), Willie Walker, Paul Lamere, Philip Kwok
14. ^ Java SE - Licensees (http://java.sun.com/javase/licensees.jsp)
15. ^ James Niccolai (January 23, 2001). "Sun, Microsoft settle Java lawsuit (http://www.javaworld.com/javaworld/jw-01-2001/jw-0124-iw-mssuncourt.html)". *JavaWorld* (IDG). http://www.javaworld.com/javaworld/jw-01-2001/jw-0124-iw-mssuncourt.html. Retrieved on 2008-07-09.
16. ^ NullPointerException (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/NullPointerException.html)
17. ^ Exceptions in Java (http://www.artima.com/designtechniques/exceptions.html)
18. ^ Using the applet Tag (The Java Tutorials > Deployment > Applets) (http://java.sun.com/docs/books/tutorial/deployment/applet/applettag.html)
19. ^ Deploying Applets in a Mixed-Browser Environment (The Java Tutorials > Deployment > Applets) (http://java.sun.com/docs/books/tutorial/deployment/applet/mixedbrowser.html)

References

- Jon Byous, *Java technology: The early years* (http://java.sun.com/features/1998/05/birthday.html). Sun Developer Network, no date [ca. 1998]. Retrieved April 22, 2005.
- James Gosling, *A brief history of the Green project* (https://duke.dev.java.net/green/). Java.net, no date [ca. Q1/1998]. Retrieved April 29, 2007.
- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java language specification*, third edition. Addison-Wesley, 2005. ISBN 0-321-24678-0 (see also online edition of the specification (http://java.sun.com/docs/books/jls/index.html)).
- Tim Lindholm and Frank Yellin. *The Java Virtual Machine specification*, second edition. Addison-Wesley, 1999. ISBN 0-201-43294-3 (see also online edition of the specification (http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html)).

External links

- Java home page (http://www.java.com/)
- Java for developers (http://java.sun.com/)
- Java Language Specification 3rd Edition (http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html)
- Java SE 6 API Javadocs (http://java.sun.com/javase/6/docs/api/)

- A Brief History of the Green Project (https://duke.dev.java.net/green/)
- Java: The Inside Story (http://sunsite.uakom.sk/sunworldonline/swol-07-1995/swol-07-java.html)
- Java Was Strongly Influenced by Objective-C (http://cs.gmu.edu/~sean/stuff/java-objc.html)
- The Java Saga (http://www.wired.com/wired/archive/3.12/java.saga.html)
- A history of Java (http://ei.cs.vt.edu/~wwwbtb/book/chap1/java_hist.html)
- The Long Strange Trip to Java (http://www.blinkenlights.com/classiccmp/javaorigin.html)
- M254 Java Everywhere (http://computing.open.ac.uk/m254/) (free open content documents from the Open University (http://www.open.ac.uk/))
- List of programming languages for a Java Virtual Machine (http://www.is-research.de/info/vmlanguages/)

Retrieved from "http://en.wikipedia.org/wiki/Java_(programming_language)"
Categories: Java programming language | Java platform | Java specification requests | C programming language family | Sun Microsystems | Concurrent programming languages | Class-based programming languages | Object-oriented programming languages | JVM programming languages | Curly bracket programming languages | Cross-platform software | Programming languages created in 1995
